Journal of Universal Computer Science, vol. 9, no. 3 (2003), 223-247
submitted: 26/6/02, accepted: 13/3/03, appeared: 28/3/03

VOC: A Methodology for the Translation Validation of Optimizing Compilers

Lenore Zuck
New York University, United States
zuck@cs.nyu.edu

Amir Pnueli
Weizmann Institute of Science, Israel and New York University, United States
amir@cs.nyu.edu

Yi Fang
New York University, United States
yifang@cs.nyu.edu

Benjamin Goldberg
New York University, United States
goldberg@cs.nyu.edu

Abstract: There is a growing awareness, both in industry and academia, of the crucial role of formally verifying the translation from high-level source-code into low-level object code that is typically performed by an optimizing compiler. Formally verifying an optimizing compiler, as one would verify any other large program, is not feasible due to its size, ongoing evolution and modification, and, possibly, proprietary considerations. *Translation validation* is a novel approach that offers an alternative to the verification of translators in general and compilers in particular: Rather than verifying the compiler itself, one constructs a validation tool which, after *every* run of the compiler, formally confirms that the target code produced in the run is a correct translation of the source program. The paper presents VOC, a methodology for the translation validation of optimizing compilers. We distinguish between *structure preserving* optimizations, for which we establish a simulation relation between the source and target code based on computational induction, and *structure modifying* optimizations, for which we develop specialized "permutation rules". The paper also describes VOC-64—a prototype translation validator tool that automatically produces verification conditions for the global optimizations of the SGI Pro-64 compiler.
Key Words: translation validation, optimizing compilers, SGI Pro-64, VOC-64, global optimizations, verification conditions, permutation rules
Category: D2.4, D3.4, I6.4

1 Introduction

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of safety-critical portions of systems. Most verification methods focus on verification of specification with respect to requirements, and high-level code with respect to specification. However, if one is to prove that the high-level specification is correctly implemented in low-level code, one needs to verify the compiler which performs the translations. Verifying the correctness of modern optimizing compilers is challenging because of the complexity and reconfigurability of the target architectures, as well as the sophisticated analysis and optimization algorithms used in the compilers.

Formally verifying a full-fledged optimizing compiler, as one would verify any other large program, is not feasible, due to its size, evolution over time, and, possibly, proprietary considerations. *Translation Validation* is a novel approach that offers an alternative to the verification of translators in general and of compilers in particular. Using the translation validation approach, rather than verify the compiler itself one constructs a *validating tool* which, after every run of the compiler, formally confirms that the target code produced is a correct translation of the source program.

The introduction of new families of microprocessor architectures, such as the EPIC family exemplified by the Intel IA-64 architecture, places an even heavier responsibility on optimizing compilers. Compile-time dependence analysis and instruction scheduling is required to exploit instruction-level parallelism in order to compete with other architectures, such as the super-scalar class of machines where the hardware determines dependences and reorders instructions at run-time. As a result, a new family of sophisticated optimizations have been developed and incorporated into compilers targeted at EPIC architectures.

Prior work ([PSS98a]) developed a tool for translation validation, CVT, that succeeded in automatically verifying translations involving approximately 10,000 lines of source code in about 10 minutes. The success of CVT critically depends on some simplifying assumptions that restrict the source and target to programs with a single external loop, and assume a very limited set of optimizations.

Other approaches [Nec00, RM00] considered translation validation for less restrictive languages allowing, for example, nested loops. They also considered a more extensive set of optimizations. However, the methods proposed there were restricted to *structure preserving* optimizations, and could not directly deal with more aggressive optimizations such as *loop distribution* and *loop tiling* that are often used in more advanced optimizing compilers.

Our ultimate goal is to develop a methodology for the translation validation of advanced optimizing compilers, with an emphasis on EPIC-targeted compilers and the aggressive optimizations characteristic to such compilers. Our methods will handle an extensive set of optimizations and can be used to implement fully automatic certifiers for a wide range of compilers, ensuring an extremely high level of confidence in the compiler in areas, such as safety-critical systems and compilation into silicon, where correctness is of paramount concern.

1.1 Overview

In this paper, we describe the initial steps towards our goal:

- the development of the theory of a *Correct translation*;
- a general proof rule for translation validation of *structure preserving* optimizations;
- VOC-64, a tool that implements the proof rule for the IR (Intermediate Representation) to IR translation of an EPIC compiler (the SGI Pro-64);
- a proof rule for dealing with many *structure modifying* transformations;

Section 2 describes our formal model and theory of correct translation. In general terms, we first give common semantics to the source and target languages using the formalism of *Transition Systems* (TS's). The notion of a target code T being a correct implementation of a source code S is then defined in terms of *refinement*, stating that every computation of T corresponds to some computation of S with matching values of the corresponding variables. In Figure 1 we present the process of refinement as completion of a mapping diagram.

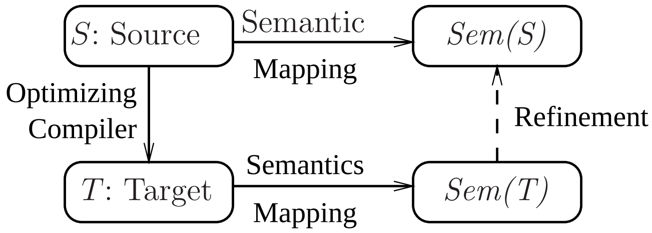


Figure 1: Refinement Completes the Picture

We distinguish between *structure preserving* optimizations, which admit a clear mapping of control points in the target program to corresponding control points in the source program, and *structure modifying* optimizations that admit no such mapping.

Structure Preserving optimizations, which cover most high-level optimizations, are the focus of Section 3. Our general approach for dealing with these optimizations is to establish a correspondence between the target and source code, based on *refinement*, and to prove it by *simulation*. According to this approach, we establish a *refinement mapping* indicating how the relevant source variables correspond to the target variables or expressions. The proof is then broken into a set of *verification conditions* (also called *proof obligations*), each claiming that a segment of target execution corresponds to a segment of source execution. In some of the cases, the proof obligations are not valid by themselves, and thus it is necessary to introduce *auxiliary invariants* which provably hold at selected points in the program. The proof obligations are then shown to be valid under the assumption of the auxiliary invariants.

For some optimizations the validating tool needs additional information that specifies which optimizing transformations have been applied in the current translation. This additional information can either be provided by the compiler or inferred by a set of heuristics and analysis techniques (some described in Section 4.) The refinement proof rule we present in Section 3 uses *invariant assertions*, which the validation tool generates at selected control points using a set of heuristics. The verification conditions are then be augmented by these invariants.

In Section 4 we describe VOC-64, our prototype tool that generates, *fully automatically*, the verification conditions for all global optimizations of the SGI Pro-64 compiler. VOC-64 builds on the theory developed in Section 3, while using heuristics to generate the correct program annotations that allow for the construction of the auxiliary invariants. The verification conditions produced by VOC-64 can be sent to CVT [PSS98a] to be verified.

A more challenging category of optimizations that is not covered in Section 3 is that of structure modifying optimizations and includes, e.g., *loop distribution* and *fusion*, *loop tiling*, and *loop interchange*. For this class, it is often impossible to apply the refinement-based rule since there are often no control points where the states of the source and target programs can be compared. We identify a large class of these optimizations, namely the *reordering transformations*, and present in Section 5 *permutation rules* that allow for their effective translation validation. We then show in Section 6 that, while loop unrolling falls naturally into the category of reordering transformation and can be dealt with by the permutation rule, it can also be dealt with by the structure preserving methodology of Section 3.

One of the side-products we anticipate from this work is the formulation of validation-oriented instrumentation, which will instruct writers of future compilers how to incorporate into the optimization modules appropriate additional outputs which will facilitate validation. This will lead to a theory of construction of *self-certifying* compilers.

1.2 Related Work

The work here is an extension of the work in [PSS98a]. The work in [Nec00] covers some important aspects of our work. For one, it extends the source programs considered from single-loop programs to programs with arbitrarily nested loop structure. An additional important feature is that the method requires no compiler instrumentation at all, and applies various heuristics to recover and identify the optimizations performed and the associated refinement mappings. The main limitation apparent in [Nec00] is that, as is implied by the single proof method described in the report, it can only be applied to structure-preserving optimizations. In contrast, our work can also be applied to structure-modifying optimizations, such as the ones associated with aggressive loop optimizations, which are a major component of optimizations for modern architectures.

The notion of correct translation that appears in [GS99] is similar to ours; however, the work there does not deal with optimizations.

Another related work is [RM00] which proposes a comparable approach to translation validation, where an important contribution is the ability to handle pointers in the source program. However, the method proposed there assumes *full* instrumentation of the compiler, which is not assumed here or in [Nec00].

More weakly related are the works reported in [Nec97] and [NL98], which do not purport to establish full correctness of a translation but are only interested in certain "safety" properties. However, the techniques of program analysis described there are very relevant to the automatic generation of refinement mappings and auxiliary invariants. Rival [Riv03] presents a methodology based on abstract-interpretation for certification of assembly code that uses the analysis of the source code and the debugging information.

The work in [Fre02] presents a framework for describing global optimizations by rewrite rules with CTL formulae as side conditions, which allow for generation of correct optimizations, but not for verification of (possibly incorrect) optimizations. The work in [GGB02] proposes a method for deploying optimizing code generation while correct translation between input program and code. They focus on code selection and instruction scheduling for SIMD machines.

Somewhat similar to our approach, in the treatment of structure modifying transformation, is the work in [SBCJ02]. There, static analysis is used to extract geometric model, on which loop data reuse transformations are checked to preserve semantics. preserving equivalence conditions can be checked that . However, this work focuses on source-to-source transformations that are manually applied in embedded system design, while the focus of our work is optimizations that are automatically generated by compilers.

2 The Model

The compiler receives a *source program* written in some high-level language, translates it into an *Intermediate Representation (IR)*, and then applies a series of optimizations to the program – starting with classical architecture-independent *global* optimizations, and then architecture-dependent ones such as register allocation and instruction scheduling. Typically, these optimizations are performed in several passes (up to 15 in some compilers), where each pass applies a certain type of optimization. In this section we briefly describe the intermediate code, which we assume to be the input and output language of the optimization phase of the compiler, and the transition system model which is our formal model.

2.1 Intermediate Code

The intermediate code is a three-address code. It is described by a *flow graph*, which is a graph representation of the three-address code. Each node in the flow graph represents a *basic block*, that is, a sequence of statements that is executed in its entirety and contains no branches. The edges of the graph represent the flow of control.

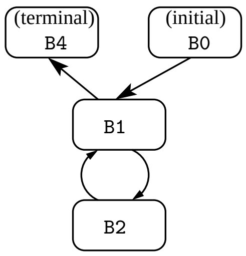
Example 1 Fig. 2 shows some C code, its translation by the SGI Pro-64 into intermediate code, and the corresponding flow graph. The code computes the integer square root of `n`.

```
int n, y, w;
n = 500;
y = 0;
w = 1;
while (w <= n)
{
  w = w+2*y+3;
  y = y+1;
}
```

(a) Source Code

```
B0  n <- 500
    y <- 0
    w <- 1
B1  WHILE (w <= n)
B2  BLOCK
    w <- w+2*y+3
    y <- y + 1
    END_BLOCK
B4
```

(b) Intermediate Representation



(c) Flowgraph

Figure 2: Source Code, IR, and Flow Graph

2.2 Transition Systems

In order to present the formal semantics of source and intermediate code we introduce *transition systems*, TS's, a variant of the *transition systems* of [PSS98b]. A *Transition System* $S = \langle V, \mathcal{O}, \Theta, \rho \rangle$ is a state machine consisting of:

- V a set of *state variables*,
- $\mathcal{O} \subseteq V$ a set of *observable variables*,

– Θ an *initial condition* characterizing the initial states of the system, and

– ρ a *transition relation*, relating a state to its possible successors.

The variables are typed, and a *state* of a TS is a type-consistent interpretation of the variables. For a state s and a variable $x \in V$, we denote by $s[x]$ the value that s assigns to x. The transition relation refers to both unprimed and primed versions of the variables, where the primed versions refer to the values of the variables in the successor states, while unprimed versions of variables refer to their value in the pre-transition state. Thus, e.g., the transition relation may include "$y' = y + 1$" to denote that the value of the variable y in the successor state is greater by one than its value in the old (pre-transition) state.

The observable variables are the variables we care about, where we treat each I/O device as a variable, and each I/O operation removes/appends elements to the corresponding variable. If desired, we can also include among the observables the history of external procedure calls for a selected set of procedures. When comparing two systems, we will require that the observable variables in the two systems match.

A computation of a TS is a maximal finite or infinite sequence of states $\sigma : s_0, s_1, \ldots$, starting with a state that satisfies the initial condition such that every two consecutive states are related by the transition relation. I.e., $s_0 \models \Theta$ and $\langle s_i, s_{i+1} \rangle \models \rho$ for every i, $0 \leq i + 1 < |\sigma|$[1].

Example 2 We translate the intermediate code in Fig. 2 into a TS. The set of state variables V includes the observables `n` and `y`, and the local variable `w`. We also include in V the control variable (program counter) π that points to the next statement to be executed. The range of π is $\{\mathtt{B0}, \mathtt{B1}, \mathtt{B2}, \mathtt{B4}\}$. The initial condition, given by Θ: $\pi = \mathtt{B0}$, states that the program starts at location (i.e. block) `B0`. As observables, we take $\mathcal{O} = \{\mathtt{n}, \mathtt{y}\}$.

The transition relation ρ can be presented as the disjunction of four disjuncts $\rho \;=\; \rho_{01} \;\vee\; \rho_{12} \;\vee\; \rho_{21} \;\vee\; \rho_{14}$, where ρ_{ij} describes all possible moves from `Bi` to `Bj` without passing through intermediate blocks.

For example, ρ_{21} is:

$$(\pi = 2) \;\wedge\; (\mathtt{w}' = \mathtt{w} + 2 * \mathtt{y} + 3) \;\wedge\; (\mathtt{y}' = \mathtt{y} + 1) \;\wedge\; (\pi' = 1)$$

When describing a transition relation, we mention only variables whose values are changed. Thus, we omit from ρ_{21} the clause $\mathtt{n}' = \mathtt{n}$ since `n` is not changed in the transition.

A computation of the program starts with `B0`, continues to `B1`, then cycles to `B2` and back to `B1` several times, and finally terminates at `B4`. The state reached at each block is described by the values assigned to the variables.

[1] $|\sigma|$, the *length of* σ, is the number of states in σ. When σ is infinite, its length is ω.

A transition system $\mathcal{T}$ is called *deterministic* if the observable part of the initial condition uniquely determines the rest of the computation. That is, if $\mathcal{T}$ has two computations $s_0, s_1, \ldots$ and $t_0, t_1, \ldots$ such that the observable part (values of the observable variables) of s_0 agrees with the observable part of t_0, then the two computations are identical. We restrict our attention to deterministic transition systems and the programs which generate such systems. Thus, to simplify the presentation, we do not consider here programs whose behavior may depend on additional inputs which the program reads throughout the computation. It is straightforward to extend the theory and methods to such intermediate input-driven programs.

The translation of an intermediate code into a TS is straightforward; we therefore assume that all code we are dealing with here is described by a TS.

2.3 Comparison and Refinement between TSs

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ be two TS's, to which we refer as the *source* and *target* TS's, respectively. Such two systems are called *comparable* if there exists a one-to-one correspondence between the observables of P_S and those of P_T. To simplify the notation, we denote by $X \in \mathcal{O}_S$ and $x \in \mathcal{O}_T$ the corresponding observables in the two systems. A source state s is defined to be *compatible* with the target state t, if s and t agree on their observable parts. That is, $s[X] = t[x]$ for every $x \in \mathcal{O}_T$. We say that P_T is a *correct translation* (*refinement*) of P_S if they are comparable and, for every $\sigma_T : t_0, t_1, \ldots$ a computation of P_T and every $\sigma_S : s_0, s_1, \ldots$ a computation of P_S such that s_0 is compatible with t_0, σ_T is terminating (finite) iff σ_S is and, in the case of termination, their final states are compatible.

Our goal is to provide an automated method that will establish (or refute) that a given target code correctly implements a given source code, where both are expressed as TSs.

3 VOC: Structure-Preserving Transformations

We use the term "structure preserving" transformation without giving it a rigorous definition. Roughly speaking, we use to the term to cover all transformations that can be dealt with by Floyd-like methodologies, i.e., structure preserving transformation allow the assignment of control points where the values of observable variables can be matched, such that each loop includes at least one control point. These transformation cover most of the global optimizations (including loop-invariant code motion) and do not cover loop transformation such as interchange, reversal, and tiling.

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ be comparable TSs, where P_S is the *source* and P_T is the *target*. In order to establish that P_T is a correct translation of P_S for the cases that P_T is derived from P_S by structure preserving transformations, we introduce a proof rule, VALIDATE. The rule is inspired by the computational induction approach ([Flo67]), originally introduced for proving properties of a single program. Rule VALIDATE provides a proof methodology by which one can prove that one program *refines* another. This is achieved by establishing a *control mapping* from target to source locations, a *data abstraction* mapping from source to target variables, and proving that these abstractions are maintained along basic execution paths of the target program.

The proof rule is presented in Fig. 3. There, each TS is assumed to have a *cut-point set* CP. This is a set of blocks that includes the initial and terminal block, as well as at least one block from each of the cycles in the programs' control flow graph. A *simple path* is a path connecting two cut-points, and containing no other cut-point as an intermediate node. For each simple path leading from `Bi` and `Bj`, ρ_{ij} describes the transition relation between blocks `Bi` and `Bj`. Typically, such a transition relation contains the condition which enables this path to be traversed, and the data transformation effected by the path. Note that, when the path from `Bi` to `Bj` passes through blocks that are not in the cut-point set, ρ_{ij} is a compressed transition relation that can be computed by the composition of the intermediate transition relation on the path from `Bi` to `Bj`.

The invariants φ_i in part (2) are program annotations that are expected to hold whenever execution visits block `Bi`. They often can be derived from the data flow analysis carried out by an optimizing compiler. Intuitively, their role is to carry information in between basic blocks. Note that we allow the data abstraction α in part (3) to be partial, and to include guards (the p_is.) The motivation for allowing α to be partial is to accommodate situations which occur, for example, in dead code elimination, where source variables have no corresponding target variables. The motivation for allowing α to contain guards is to accommodate situations occurring, for example, in loop invariant code motion where, at some points of the execution, source variables have no correspondence in the target while at other points they do. The guards, thus, describe the conditions under which the source variables can be defined using target variables.

The verification conditions assert that at each (target) transition from `Bi` to `Bj`[2], if the assertion φ_i and the data abstraction hold before the transition, and the transition takes place, then after the transition there exist new source variables that reflect the corresponding transition in the source, and the data abstraction and the assertion φ_j hold in the new state. Hence, φ_i is used as a hypothesis at the antecedent of the implication C_{ij}. In return, the validator also has to establish that φ_j holds after the transition. Thus, as part of the verification effort, we confirm that the proposed assertions are indeed inductive and hold whenever the corresponding block is visited. Since the assertion mentions only

[2] Recall that we assume that a path described by the transition is simple.

1. Establish a *control abstraction* κ: $\mathsf{CP}_T \rightarrow \mathsf{CP}_S$ that maps initial and terminal blocks of the target into the initial and terminal blocks of the source.
2. For each basic block `Bi` in CP_T, form an *invariant* φ_i that may refer only to concrete (target) variables.
3. Establish a *data abstraction*

$$\alpha : (p_1 \rightarrow V_1 = e_1) \ \wedge \ \cdots \ \wedge \ (p_n \rightarrow V_n = e_n)$$

assigning to *some* non-control source state variables $V_i \in V_S$ an expression e_i over the target state variables, conditional on the (target) boolean expression p_i. Note that α may contain more than one clause for the same variable. It is required that, for every *observable* source variable $V \in \mathcal{O}_S$ (whose target counterpart is v) and every terminal target block `B,` α implies that $V = v$ at `B.`
4. For each pair of basic blocks `Bi` and `Bj` such that there is a simple path from `Bi` to `Bj` in the control graph of P_T, let $Paths(\kappa(i), \kappa(j))$ be the set of simple source paths connecting block $\mathtt{B}\kappa(\mathtt{i})$ to $\mathtt{B}\kappa(\mathtt{j})$. We form the verification condition

$$C_{ij} : \qquad \varphi_i \wedge \alpha \wedge \overset{T}{\rho}_{ij} \quad \rightarrow \quad \exists {V_S}' : (\bigvee_{\pi \in Paths(\kappa(i), \kappa(j))} \overset{S}{\rho}_{\pi}) \wedge \alpha' \wedge \varphi'_j,$$

where $\overset{S}{\rho}_{\pi}$ is the transition relation for the simple source path π.
5. Establish the validity of all the generated verification conditions.

Figure 3: The Proof Rule VALIDATE

target variables, their validity should depend solely on the target code.

In most cases, the primed source variables can be easily determined from the code, and the existential quantification in verification condition (4) can be eliminated. This is because, in the case that E contains no free occurrences of the variable x', the implication $q \rightarrow \exists x' : (x' = E \ \wedge \ r)$ is validity-equivalent to the implication $q \ \wedge \ (x' = E) \rightarrow r$. However, this may not always be the case. In some transformations there is no bijection between the different paths connecting two control points in the source and the paths connecting the corresponding (via κ^{-1}) control points in the target. Another case where the identification of the primed source variables is not easily determined from the code is when the data abstraction α is partial. We present below an example for such cases. We are therefore forced to leave the existential quantifier in (4).

In Appendix A we prove the soundness of the rule. In Section 4 we describe VOC-64 – a tool, currently being developed at NYU, to automatically generate verification conditions (VCs) for programs being compiled by the SGI Pro-64 compiler. Verification of the VCs that are generated by VOC-64 can be performed by CVT, which was developed for the *Sacres* project [PRSS99]. The parts that cannot be handled by CVT can be handled by other theorem provers. We have been using STeP [MAB+94] and are exploring other packages that can provide similar capabilities.

Before we describe voc-64 and its capabilities, we give an example of an application of VALIDATE as generated by voc. For readability, the output here is slightly edited, but is essentially as produced by the tool. In the next section, we describe the tool in more detail.

Examples of the Application of VALIDATE

Our first example is derived using voc-64 which is described in Section 4. Here we sketch some of the relevant outputs. Consider the program of Fig. 2 after a series of optimizations: Constant folding, copy propagation, dead code elimination, control flow graph optimization (loop inversion), and strength reduction. A simplified version of the resulting code, where we renamed some variables, is in Fig. 4. The annotation (φ_1, denoted `phi1`) is supplied by the compiler (see Section 4).

```
B0 N <- 500
   Y <- 0
   W <- 1
B1 WHILE (W <= N)
B2 BLOCK
   W <- W + 2 * Y + 3
   Y <- Y + 1
   END_BLOCK
B4
```

(a) Input Program

```
B0 .t264 <- 0
   y <- 0
   w <- 1
B1 {phi1: .t264 = 2 * y}
   w <- .t264 + w + 3
   y <- y + 1
   .t264 <- .t264 + 2
   IF (w <= 500) GOTO B1
B2
```

(b) Optimized Code

Figure 4: Example 1: Source and Annotated Target Programs

To validate the program, we use the control mapping $\kappa = \{0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 4\}$, and the data abstraction

$$\alpha: \begin{pmatrix} (\mathtt{PC} = \kappa(\mathtt{pc})) \ \wedge \ (\mathtt{pc} \neq 0 \rightarrow \mathtt{Y} = \mathtt{y}) \ \wedge \\ (\mathtt{pc} \neq 0 \rightarrow \mathtt{W} = \mathtt{w}) \ \wedge \ (\mathtt{pc} \neq 0 \rightarrow \mathtt{N} = 500) \end{pmatrix}$$

where variable names in upper case denote source (abstract) variables, and lower case letters denote their target (concrete) counterparts. Note that we always include in α the control mapping $\mathtt{PC} = \kappa(\mathtt{pc})$.

The verification condition C_{01} obtained for the simple path from `B0` to `B1`, after simplification (including the removal of the existential quantifier), is:

$$C_{01}: \quad \alpha \ \wedge \ \overset{T}{\rho}_{01} \ \wedge \ \alpha' \ \rightarrow \ \overset{S}{\rho}_{02} \ \wedge \ \varphi_1'$$

where $\overset{T}{\rho}_{01}$ is defined by:

$$\big((\mathtt{pc} = 0) \ \wedge \ (.t264' = 0) \ \wedge \ (\mathtt{y}' = \mathtt{0}) \ \wedge \ (\mathtt{w}' = \mathtt{1}) \ \wedge \ (\mathtt{pc}' = \mathtt{1})\big)$$

and $\overset{S}{\rho}_{02}$ is defined by:

$$\big((\mathtt{PC} = 0) \ \wedge \ (\mathtt{Y}' = 0) \ \wedge \ (\mathtt{W}' = 1) \ \wedge \ (\mathtt{N}' = 500) \ \wedge \ (\mathtt{N}' \geq \mathtt{W}') \ \wedge \ (\mathtt{PC}' = 2)\big)$$

We also have:

$$\alpha': \ \big((\mathtt{PC}' = \kappa(\mathtt{pc}')) \ \wedge \ (\mathtt{Y}' = \mathtt{y}') \ \wedge \ (\mathtt{W}' = \mathtt{w}') \ \wedge \ (\mathtt{N}' = \mathtt{500})\big)$$

and φ_1': $(.t264' = 2 * \mathtt{y}')$. The other verification conditions are constructed similarly. They are all trivial to verify.

Our second example demonstrates the need for the existential quantifier in the VC. Consider the following source and target programs:

```
B0 I <- 0
B1 X <- A + B
   IF !(I=0) GOTO B3
B2 I <- X
   GOTO B4
B3 I <- I + X
B4 Y <- A + I
   IF (I<100) GOTO B1
B5
```

(a) Input Program

```
B0 i <- 0
   x <- a + b
B1 {phi: x = a + b}
   i <- i + x
   if (i < 100) goto B1
B2 y <- a + i
B3
```

(b) Optimized Code

Figure 5: Example 2: Source and Annotated Target Programs

To validate this transformation using VALIDATE, we use the control mapping $\kappa = \{0 \mapsto 0, 1 \mapsto 1, 3 \mapsto 5\}$, and the data abstraction

$$\alpha: \ (\mathtt{PC} = \kappa(\mathtt{pc})) \ \wedge \ \mathtt{I} = \mathtt{i} \ \wedge \ \mathtt{A} = \mathtt{a} \ \wedge \ \mathtt{B} = \mathtt{b} \ \wedge \ (\mathtt{pc} \neq \mathtt{1} \rightarrow \mathtt{X} = \mathtt{x} \ \wedge \ \mathtt{Y} = \mathtt{y})$$

The verification condition C_{11} obtained for the simple path from `B1` to `B1` requires computing $\overset{S}{\rho}_{11}$, which is a disjunction of the transition relations of the path $\mathtt{B1} \rightarrow \mathtt{B3} \rightarrow \mathtt{B4} \rightarrow \mathtt{B1}$ and the path $\mathtt{B1} \rightarrow \mathtt{B2} \rightarrow \mathtt{B4} \rightarrow \mathtt{B1}$. Consequently, we obtain

$$\rho^{S}_{11} = \mathtt{PC} = 1 \ \wedge \ \mathtt{PC}' = 1 \ \wedge \left(\begin{array}{l} (I = 0 \ \wedge \ A + B < 100 \ \wedge \ I' = A + B \ \wedge \ X' = A + B \ \wedge \\ \ Y' = 2A + B \ \wedge \ A' = A \ \wedge \ B' = B) \ \vee \\ (I \neq 0 \ \wedge \ I + A + B < 100 \ \wedge \ \ I' = I + A + B \ \wedge \ X' = A + B \ \wedge \\ \ Y' = I + 2A + B \ \wedge \ A' = A \ \wedge \ B' = B) \end{array} \right)$$

While for ρ^{T}_{11} we have:

$$\rho^{T}_{11} = (i + x < 100 \ \wedge \ i' = i + x \ \wedge \ x' = x \ \wedge \ y' = y \ \wedge \ a' = a \ \wedge \ b' = b)$$

Thus, the verification condition C_{11} is $\varphi_1 \ \wedge \ \alpha \ \wedge \ \rho^{T}_{11} \rightarrow \exists I', X', Y', A', B' : \varphi'_1 \ \wedge \ \alpha' \ \wedge \ \rho^{S}_{11}$. Here, the existential quantifier cannot be immediately removed, since there are two possible paths in the source, both implying different values for the primed source variables. (This, of course, assumes the validator cannot attempt to reconcile the two branches on its own without a theorem provers)

4 VOC-64: A translation Validator for the SGI Pro-64 Compiler

This section presents an overview of the components of VOC-64 : the parser and the generators of cut-point sets, simple paths, transition relations, invariants, data abstraction, control mapping, and verification conditions. Thus, VOC-64 performs its own control- and data-flow analyses, providing a cross-check of the analyses of the compiler. The description of the components includes examples of the tool's output using the source program of Fig. 2 and its target of Fig. 4. The documentation for VOC-64 and examples of complete runs are in *http://www.cs.nyu.edu/validation/tvoc*.

Parser (*wh.cxx* and *wh.h*). The intermediate language of the SGI Pro-64 compiler is WHIRL. After each round of optimization, the compiler outputs ASCII formatted WHIRL code, which can be read by a parser and translated back into a graphic representation.

Cut-Point Set(*ts_more.cxx*). VOC-64 computes cut-point sets, `CP_SET`, for both source and target as follows. The cut-point sets include initial and terminal blocks. For each loop, the first block loop's body will be chosen as cut-point.

For the example of the previous section, VOC-64 computes the cut-point set $\{0, 2, 4\}$ for the source, and $\{0, 1, 2\}$ for the target.

Control Mapping (*ts_more.cxx*) VOC-64 maps each initial and terminal location of the target to initial and terminal locations of the source.

Since we are dealing with structure preserving optimization, it is not hard to establish one-to-one correspondence between loops in the target and loops in the source. Thus each cut-point of a target loop is mapped to the cut-point of the corresponding source loop.

For our example, VOC-64 produces {`0 -> 0, 1 -> 2, 2 -> 4`}.

Paths (*ts_more.cxx*). VOC-64 computes sets `CP_PATHS` of paths for each source and target. The set includes all *simple* paths (and cycles) between two points in the appropriate `CP_SET`, i.e., paths that do not contain as intermediate points any other point in `CP_SET`.

For our example, VOC-64 computes (for the source)
$\mathtt{CP\_PATHS} = \{\mathtt{B0} \rightarrow \mathtt{B1} \rightarrow \mathtt{B2},\ \mathtt{B2} \rightarrow \mathtt{B1} \rightarrow \mathtt{B2},\ \mathtt{B0} \rightarrow \mathtt{B1} \rightarrow \mathtt{B4},\ \mathtt{B2} \rightarrow \mathtt{B1} \rightarrow \mathtt{B4}\}$

Transition Relation For each path $\mathtt{Bi} \rightarrow \mathtt{Bj}$ in `CP_PATHS`, VOC-64 computes the τ_{ij}, which is ρ_{ij} without the control information (which is implicit) The output is `Ps(i,j)` for source, and `Pt(i,j)` for target, which consists of a set of equalities over the appropriate variables and the branch conditions. Thus, ρ^x_{ij} is a conjunct of the terms in `Px(i,j)`, together with $\pi = i \wedge \pi' = j$. For example, for `Ps(0,2)` VOC-64 produces

```
(N = 500) & (Y = 0) & (W = 1) & (1 <= 500)
```

Invariants (*ts.cxx*). VOC-64 computes two types of invariant assertions for each basic block. One type is derived from the reachable definitions (obtained by data flow analysis). These invariants help VOC-64 handle optimizations as copy propagation and code motion. The other type is derived from loop induction variables. These invariants help VOC-64 handle optimizations such as strength reduction.

In Appendix B we describe how VOC-64 computes invariants of the first type. However, we observed that the invariants so derived cannot handle strength reduction, which led us to generate invariants that build on the loop induction variables. This is accomplished by computing, for loop induction variables i and j whose initial values are i_0 and j_0 respectively, and that get incremented in each iteration by c_1 and c_2 respectively, the invariant $j = c_2/c_1 * i + (j_0 - i_0 * c_2/c_1)$.

Data Abstraction (*ts_more.cxx*). The data abstraction that VOC-64 computes is a set (conjunction) of implications of the form $\mathtt{pc} = i \rightarrow (\mathtt{v} = \mathtt{V})$). These implications are computed for each basic block `Bi`. That is, VOC-64 computes, for each target block $\mathtt{B}(i)$, equalities of the type $v = V$. Let $\mathtt{alpha}(i)$ denote the equalities for target block `Bi`. For each target block `Bj` that leads into `Bi`, VOC-64 computes the set of such equalities that are preserved in the path from `Bj` into `Bi`, so that $\mathtt{alpha}(i)$ is the intersection of all such sets from all incoming paths. The computation of $\mathtt{alpha}(i)$ is described in Fig. 6.

```
repeat for every path from Bj to Bi
       beta(j,i) := {(v = V) s.t.
```
$$\varphi_i \;\wedge\; \alpha \;\wedge\; \overset{T}{\rho_{ji}} \;\wedge\; \overset{S}{\rho_{\kappa(j)\kappa(i)}} \rightarrow (v = V)\}$$
```
       alpha(i)  := intersection beta(j,i)
                         for all paths from Bj to Bi
until sets stabilize
```

Figure 6: Computation of alpha(i)

The computation is successful when each target path has a unique source path that maps to it. Currently, VOC-64 does not handle data abstraction for cases when a single target path maps into multiple source paths.

In our example, we obtain in the data abstraction, both $(\mathtt{pc} = 1) \rightarrow (\mathtt{y} = \mathtt{y})$ and $(\mathtt{pc} = 2) \rightarrow (\mathtt{y} = \mathtt{y})$.

Generation of Verification Conditions The construction of the VCs is usually straightforward. An interesting case is, however, when there are multiple paths connecting two cut-points in both source and target. Assume that the cut-point set includes Bi and Bj. Assume further that there are m different paths from Bi to Bj, each contributing a disjunct P_k to $\rho^T_{i,j}$, and that there are n different paths from Bκ(i) to Bκ(j), each contributing a disjunct Q_ℓ to $\rho^S_{\kappa(i),\kappa(j)}$. Then VOC-64 generates m VCs C^k_{ij}, $k = 1,\ldots,m$, each of the form $\varphi_i \;\wedge\; \alpha \;\wedge\; P_k \quad\rightarrow\quad \exists {V_S}' \colon \bigvee_{\ell=1}^{n} Q_\ell \;\wedge\; \alpha' \;\wedge\; \varphi'_j$.

5 Validating Loop Reordering Transformations

A *reordering transformation* is any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statement [AK02]. It preserves a dependence if it preserves the relative execution order of the source and target of that dependence, and thus preserves the meaning of the program. Reordering transformations cover many of the loop transformations, including fusion, distribution, interchange, tiling, unrolling, and reordering of statements within a loop body.

In this section we review the reordering loop transformations and propose "permutation rules" that the validator may use to deal with these transformations.

5.1 Overview of Reordering Loop Transformations

Consider a loop of the form described in Fig. 7. Schematically, we can describe

$$
\begin{array}{l}
\textbf{for } i_1 = L_1 \textbf{ to } H_1 \textbf{ do} \\
\quad \ldots \\
\quad\quad \textbf{for } i_m = L_m \textbf{ to } H_m \textbf{ do} \\
\quad\quad\quad B(i_1, \ldots, i_m) \\
\quad\quad \textbf{end} \\
\quad \ldots \\
\textbf{end}
\end{array}
$$

Figure 7: A General Loop

such a loop in the form

$$\textbf{for } \mathbf{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\mathbf{i})$$

where $\mathbf{i} = (i_1, \ldots, i_m)$ is the list of nested loop indices, and $\mathcal{I}$ is the set of the values assumed by $\mathbf{i}$ through the different iterations of the loop. The set $\mathcal{I}$ can be characterized by a set of linear inequalities. For example, for the loop of Fig. 7, $\mathcal{I}$ is

$$\mathcal{I} \quad = \quad \{(i_1, \ldots, i_m) \mid L_1 \leq i_1 \leq H_1 \ \wedge \ \cdots \ \wedge \ L_m \leq i_m \leq H_m\}$$

The relation $\prec_{\mathcal{I}}$ is the ordering by which the various points of $\mathcal{I}$ are traversed. For example, for the loop of Fig. 7, this ordering is the lexicographic order on $\mathcal{I}$.

In general, a loop transformation has the following form:

$$\textbf{for } \mathbf{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\mathbf{i}) \quad \Longrightarrow \quad \textbf{for } \mathbf{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\mathbf{j})) \tag{1}$$

In such a transformation, we may possibly change the domain of the loop indices from $\mathcal{I}$ to $\mathcal{J}$, the names of loop indices from $\mathbf{i}$ to $\mathbf{j}$, and possibly introduce an additional linear transformation in the loop's body, changing it from the source $\mathtt{B}(\mathbf{i})$ to the target body $\mathtt{B}(F(\mathbf{j}))$.
An example of such transformation is *loop reversal* which can be described as

$$\textbf{for } i = 1 \textbf{ to } N \textbf{ do } \mathtt{B}(i) \quad \Longrightarrow \quad \textbf{for } j = N \textbf{ to } 1 \textbf{ do } \mathtt{B}(j)$$

For this example, $\mathcal{I} = \mathcal{J} = [1..N]$, the transformation F is the identity, and the two orders are given by $i_1 \prec_{\mathcal{I}} i_2 \iff i_1 < i_2$ and $j_1 \prec_{\mathcal{J}} j_2 \iff j_1 > j_2$, respectively.

Since we expect the source and target programs to execute the same instances of the loop's body (possibly in a different order), we should guarantee that the mapping $F : \mathcal{J} \mapsto \mathcal{I}$ is a bijection from $\mathcal{J}$ to $\mathcal{I}$, i.e. a 1-1 onto mapping. Often, this guarantee can be ensured by displaying the inverse mapping $F^{-1} : \mathcal{I} \mapsto \mathcal{J}$, which for every value of $\mathbf{i} \in \mathcal{I}$ provides a unique value of $F^{-1}(i) \in \mathcal{J}$.

Some common examples of transformations which fall into the class considered here are presented in Fig. 8 and Fig. 9. For each transformation, we describe the source loop, target loop, set of loop control variables for source ($\mathcal{I}$) and target ($\mathcal{J}$), their ordering ($\prec_{\mathcal{I}}$ and $\prec_{\mathcal{J}}$), and the bijection $F\colon \mathcal{J} \mapsto \mathcal{I}$. For tiling we assume that c divides n and d divides m.

	Interchange	**skewing**

Source	`for` $i_1 = 1, n$ `do`
	`for` $i_2 = 1, m$ `do`
	$B(i_1, i_2)$
		`for` $i_1 = 1, n$ `do`
		`for` $i_2 = 1, n$ `do`
		$B(i_1, i_2)$

Target	`for` $j_1 = 1, m$ `do`
	`for` $j_2 = 1, n$ `do`
	$B(j_2, j_1)$
		`for` $j_1 = 1, n$ `do`
		`for` $j_2 = j_1 + 1, j_1 + n$ `do`
		$B(j_2, j_2 - j1)$

$\mathcal{I}$	$\{1, \ldots, n\} \times \{1, \ldots, m\}$	$\{1, \ldots, n\} \times \{1, \ldots, n\}$
$\mathcal{J}$	$\{1, \ldots, m\} \times \{1, \ldots, n\}$	$\{(j_1, j_2) : 1 \leq j_1 \leq n \ \wedge\ j_1 + 1 \leq j_2 \leq j_1 + n\}$
$i < i'$	$\mathbf{i} <_{\text{lex}} \mathbf{i'}$	$\mathbf{i} <_{\text{lex}} \mathbf{i'}$
$j > j'$	$\mathbf{j} <_{\text{lex}} \mathbf{j'}$	$\mathbf{j} <_{\text{lex}} \mathbf{j'}$
$F(\mathbf{j})$	(j_2, j_1)	$(j_1, j_2 - j_1)$
$F^{-1}(\mathbf{i})$	(i_2, i_1)	$(i_1, i_1 + i_2)$

Figure 8: Some Loop Transformations

	Reversal	**Tiling**

Source	`for` $i = 1, n$ `do`
	$B(i)$
		`for` $i_1 = 1, n$ `do`
		`for` $i_2 = 1, m$ `do`
		$B(i_1, i_2)$

Target	`for` $j = n, 1$ `do` $B(j)$	`for` $j_1 = 1, n$ `by` c `for` $j_2 = 1, m$ `by` d `for` $j_3 = j_1, j_1 + c - 1$ `for` $j_4 = j_2, j_2 + d - 1$ $B(j_3, j_4)$
$\mathcal{I}$	$\{1, \ldots, n\}$	$\{1, \ldots, n\} \times \{1, \ldots, m\}$
$\mathcal{J}$	$\{1, \ldots, n\}$	$\{(j_1, j_2, j_3, j_3) : 1 \leq j_1 \leq n\ j_1 \equiv 1 \bmod c \ \wedge\ j_1 \leq j_3 < j_1 + c \ \wedge$ $1 \leq j_2 \leq m\ j_2 \equiv 1 \bmod d \ \wedge\ j_2 \leq j_4 < j_2 + d\}$
$\mathbf{i} \prec_{\mathcal{I}} \mathbf{i'}$	$i < i'$	$\mathbf{i} <_{\text{lex}} \mathbf{i'}$
$\mathbf{j} \prec_{\mathcal{J}} \mathbf{j'}$	$j > j'$	$\mathbf{j} <_{\text{lex}} \mathbf{j'}$
$F(\mathbf{j})$	j	(j_3, j_4)
$F^{-1}(\mathbf{i})$	i	$(c\lfloor \frac{i_1 - 1}{c} \rfloor + 1, d\lfloor \frac{i_2 - 1}{d} \rfloor + 1, i_1, i_2)$

Figure 9: Some Loop Transformations

5.2 Permutation Rules

The simulation-based proof method discussed in previous sections assumes that the source and target have similar structures and that the order between computation segments is essentially preserved in the translation. Since the transformations considered here may radically reorder the order between computations, we can no longer use rule VALIDATE for their validation. Therefore, we develop in this section a rule called the "permutation rule" for validating reordering transformations. The soundness of the permutation rule is established separately. Usually, structure-modifying optimizations are applied to small localized sections of the source program, while the rest of the program is only optimized by structure-preserving transformations. Therefore, the general validation of a translation will combine these two techniques.

There are two requirements we wish to establish in order to justify the transformation described in (1).

1. The mapping F is a bijection from $\mathcal{J}$ onto $\mathcal{I}$. That is, F establishes a 1-1 correspondence between elements of $\mathcal{J}$ and the elements of $\mathcal{I}$.

2. For every $\mathbf{i}_1 \prec_{\mathcal{I}} \mathbf{i}_2$ such that $F^{-1}(\mathbf{i}_1) \succ_{\mathcal{J}} F^{-1}(\mathbf{i}_2)$, $\mathtt{B}(\mathbf{i}_1); \mathtt{B}(\mathbf{i}_2) \sim \mathtt{B}(\mathbf{i}_2); \mathtt{B}(\mathbf{i}_1)$. This requirement is based on the observation that in the source computation, $\mathtt{B}(\mathbf{i}_1)$ is executed before $\mathtt{B}(\mathbf{i}_2)$ while the corresponding $\mathtt{B}(\mathbf{i}_1)$ is executed after $\mathtt{B}(\mathbf{i}_2)$ in the target computation. The overall results will be the same if all of these permutation relations hold between pairs of iterations whose order of execution is reversed between source and target.

These requirements are summarized in rule PERMUTE, presented in Fig. 10.

$$
\begin{array}{lll}
\text{R1. } \forall \mathbf{i} \in \mathcal{I} : \exists \mathbf{j} \in \mathcal{J} : & \mathbf{i} = F(\mathbf{j}) & \\
\text{R2. } \forall \mathbf{j}_1 \neq \mathbf{j}_2 \in \mathcal{J} : & F(\mathbf{j}_1) \neq F(\mathbf{j}_2) & \\
\text{R3. } \forall \mathbf{i}_1, \mathbf{i}_2 \in \mathcal{I} : & \mathbf{i}_1 \prec_{\mathcal{I}} \mathbf{i}_2 \ \wedge \ F^{-1}(\mathbf{i}_1) \succ_{\mathcal{J}} F^{-1}(\mathbf{i}_2) & \Longrightarrow \\
 & \qquad \mathtt{B}(\mathbf{i}_1); \mathtt{B}(\mathbf{i}_2) \ \sim \ \mathtt{B}(\mathbf{i}_2); \mathtt{B}(\mathbf{i}_1) & \\
\hline
\multicolumn{3}{c}{\textbf{for } \mathbf{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\mathbf{i}) \quad \sim \quad \textbf{for } \mathbf{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\mathbf{j}))}
\end{array}
$$

Figure 10: Permutation Rule PERMUTE for reordering transformations

In order to apply rule PERMUTE to a given case, it is necessary to identify the function F (and F^{-1}) and validate premises R1-R3. The identification of F can be provided to us by the compiler, once it determines which of the relevant loop optimizations it chooses to apply. Of course, the validator will need to verify that

F is a bijection and that the source and target bodies relate to each other as prescribed by the rule. Alternately, we can develop heuristics which can attempt to identify these functions based on a comparison of the source and target codes.

In most of the applications, the functions F and F^{-1} are based on linear expressions. For these cases, premises R1-R3 can be automatically validated using any implementation of linear integer arithmetic decision procedures, such as the ones contained in ICS, CVC or PVS.

The techniques presented here only dealt with transformations which reorder the execution of the entire loop's body. They can be easily generalized to deal with cases where the loop's body is partitioned into several segments, and each of the segments is moved to a different iteration. Such transformations occur in the case of loop fusion and distribution as well as loop pipelining.

6 A Note on Loop Unrolling

A schematic loop unrolling is in Fig. 11 (where we assume $c > 1$.) There are

```
L1: i = 1
L2: B(i); i=i+1;
    if (i<=n) goto L2
L3:
```

(1) Source Program

```
L1: B(1); ...; B(n mod c)
    if (n < c) goto L5
L3: i = (n mod c) + 1;
L4: B(i); B(i+1); ... ; B(i+c-1);
    i=i+c; if (i < n) goto L4
L5:
```

(b) Target Program

Figure 11: Loop Unrolling

several strategies for dealing with loop unrolling. One is to design a premutaion-like rule that deals with it directly. Another is to consider loop unrolling as a special case of tiling an $n \times 1$ array with tiles of size c, and then unrolling the innermost loop. A third approach, which we pursue here, is to consider loop unrolling as a structure-preserving transformation and apply VALIDATE to it.

The formal treatment of the loop unrolling optimization was considered first by Raya Leviathan, who has suggested the application of rule VALIDATE to the problem. As reported in [PZL01], she used the verification system STeP in order to conduct a deductive verification of the example we bring below. The reason we repeat this example here is in order to show how voc can complete this validation task in a fully automatic manner.

For a simple path between two cut-points in the target, voc finds its corresponding execution in the source and checks whether the branch conditions of the two paths are logically equivalent. To this end, we allow that the source paths used in VALIDATE include non-simple paths, bounding the number of simple paths composed to derive each non-simple path to c, where c is the unrolling constant. We demonstrate the approach on the C-code and its translation described in Fig. 12.

C Code

```
long a[100];
void unroll(int n)
{
  int i;
  for(i=0; i<n; i++)
      a[i] = i;
}
```

Source Code (IR)

```
B0  i <- 0
B1  WHILE (i < n)
B2     [&a + i * 8] <- i
       i <- i + 1
B3  RETURN
```

Target

```
B0  IF !(0 < n) GOTO B6
B1  i <- 0
    t1 <- &a
    t2 <- n MOD 3
    IF (t2 = 0) GOTO B4
B2  [t1] <- i
    t1 <- t1 + 8
    i <- i + 1
    IF (t2 = 1) GOTO B4
B3  [t1] <- i
    t1 <- t1 + 8
    i <- i + 1
B4  IF !(i < n) GOTO B6
B5  t10 <- t1
    t11 <- t1 + 8
    t12 <- t1 + 16
    t20 <- i
    t21 <- i + 1
    t22 <- i + 2
    [t10] <- t20
    [t11] <- t21
    [t12] <- t22
    t1 <- t1 + 24
    i <- i + 3
    IF (i < n)
        GOTO B5
B6  RETURN
```

Figure 12: A Loop Unrolling Example

VOC-64 generated the control mapping $\{0 \mapsto 0, 5 \mapsto 2, 6 \mapsto 3\}$, the data abstraction $N = n \ \wedge \ (\pi > 1 \rightarrow (I = i))$ and the invariants

$$\begin{aligned} &\varphi_0: \ \mathrm{T} \\ &\varphi_5: \ (t1 = 8 \cdot i + \&a) \ \wedge \ (i < n) \ \wedge \ ((n - i) \bmod 3 = 0) \\ &\varphi_6: \ (t1 = 8 \cdot i + \&a) \ \wedge \ ((n - i) \bmod 3 = 0) \end{aligned}$$

For example, for $C_{56} = \varphi_5 \ \wedge \ \alpha \ \wedge \ \overset{T}{\rho}_{56} \rightarrow \exists V'_{S} : \overset{S}{\rho}_{23} \ \wedge \ \alpha' \ \wedge \ \varphi'_6$, where $\overset{S}{\rho}_{23}$ is the non-simple path B2 $\rightarrow$ B2 $\rightarrow$ B2 $\rightarrow$ B3 which visits four cut-points (two of which are intermediate). VOC-64 generated for $\overset{T}{\rho}_{56}$:

$$\left(\begin{array}{l} (\pi = 5) \ \wedge \ (\pi' = 6) \ \wedge \ i' = i + 3) \ \wedge \ (t1' = t1 + 24) \ \wedge \ ([t1]' = i) \\ \wedge \ ([t1 + 8]' = i + 1) \ \wedge \ ([t1 + 16]' = i + 2) \ \wedge \ (i + 3 \geq n) \end{array} \right)$$

and for (non-simple) $\overset{S}{\rho}_{23}$:

$$\left(\begin{array}{l} [8 \cdot I + \&A]' = I) \ \wedge \ ([8 \cdot (I + 2) + \&A]' = I + 2) \ \wedge \\ ([8 \cdot (I + 1) + \&A]' = I + 1) \ \wedge \ (\Pi = 2) \ \wedge \ (I' = I + 3) \ \wedge \\ (I + 1 < N) \ \wedge \ (I + 2 < N) \ \wedge \ (I + 3 \geq N) \ \wedge \ (\Pi' = 3) \end{array} \right)$$

The paper [PZL01] presents a compilation of the Trimaran compiler that involves this example of loop unrolling, and verifies it using rule VALIDATE. In fact, in one case the validation failed, showing that Trimaran generated target code that could possibly cause a segmentation fault, although the source code would not have.

7 Summary and Future Work

This paper presented VOC, the theoretical framework for translation validation of optimizing compilers. We described VOC-64, a tool we are developing to perform automatic translation validation for the SGI Pro-64 compiler, and gave examples of the VCs it generates. VOC-64 currently generates verification conditions for all the classical global optimizations of the SGI Pro-64, as well as for loop unrolling.

We also presented permutation rules for dealing with reordering loop transformations. We are currently working on extending VOC to accommodate the permutation rules.

We have recently connected VOC-64 to the theorem prover ICS [FORS01], that verifies the verification condition generated by VOC-64 . We are working on extensions to the theorem prover CVC [SBD02] so that it can provide automatic proofs to the permutation rules used for loop optimizations.

Our approach has a few weaknesses. It does not check for exceptions (over- and under-flow, zero-divide, out of bound array references, etc.) These may cause both false negative and false positive validation results. Similarly, the approach does not check for exceptions that are introduced, or eliminated, during the compilation. These may only cause false negatives. Finally, the ability of our approach to check correctness of algebraic simplifications is limited by the power of the decision procedure employed by the theorem prover we use.

The work presented here does not deal with pointers, aliasing, and procedure calls. We intend to remedy this soon. We have also embarked on developing the theory for dealing with instruction scheduling and other machine-dependent optimizations, as well as on developing techniques for run-time validation of aggressive optimizations that cannot be validated by static tools.

Acknowledgements

We would like to thank Clark Barrett for many helpful discussions and comments. We would also like to thank the referees for their diligent comments on this paper.

This research was supported in part by NSF grant CCR-0098299, ONR grant N00014-99-1-0131, and the John von Neumann Minerva Center for Verification of Reactive Systems.

References

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[ASU88] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1988.

[Flo67] R.W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[FORS01] J.C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *Proc.* 13^{rd} *Intl. Conference on Computer Aided Verification (CAV'01), volume 2102 of*Lect. Notes in Comp. Sci.*, Springer-Verlag*, 2001.

[Fre02] C.C. Frederiksen. Correctness of Classical Compiler Optimizations using CTL. In *Proc. of Compiler Optimization meets Compiler Verificaiton (COCV) 2002*, Electronic Notes in Theoretical Computer Science (ENTCS), volume 65, issue 2.

[GGB02] S. Glesner, R. Geiß and B. Boesler. Verified Code Generation for Embedded Systems. In *Proc. of Compiler Optimization meets Compiler Verificaiton (COCV) 2002*, Electronic Notes in Theoretical Computer Science (ENTCS), volume 65, issue 2.

[GS99] G. Goos and W. Zimmermann. Verification of Compilers. In *Correct System Design*, volume 1710 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pages 201–230, 1999.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.

[Muc97] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[Nec97] G.C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. Princ. of Prog. Lang.*, pages 106–119, 1997.

[Nec00] G. Necula. Translation validation of an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 2000*, pages 83–95, 2000.

[NL98] G.C. Necula and P. Lee. The design and implementation of a certifying compilers. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 1998*, pages 333–344, 1998.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domains instantiations. In *In N. Halbwachs and D. Peled, editors, Proc.* 11^{st} *Intl. Conference on Computer Aided Verification (CAV'99), volume 1633 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 455–469, 1999.

[PSS98a] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)-automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[PSS98b] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *In B. Steffen, editor, Proc.* 4^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), volume 1384 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 151–166, 1998.

[PZP00] A. Pnueli, L. Zuck, and P. Pandya. Translation validation of optimizing compilers by computational induction. Technical report, Courant Institute of Mathematical Sciences, New York University, 2000.

[PZL01] A. Pnueli, L. Zuck, and R. Leviathan. Validation of Optimizing Compilers. Technical report, Computer Science Department, NYU

(CIMS/CS/NYU.) Available in `www.cs.nyu.edu/validator/pubs.html` `http://www.cs.nyu.edu/validator/pubs.html`

[Riv03] X. Rival. Abstract Interpretation-Based Certification of Assembly Code. In *Proc. 4^{th} Intl. Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pages 41–55, 2003.

[RM00] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Run-Time Result Verification Workshop*, Trento, July 2000.

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lect. Notes in Comp. Sci.* Springer-Verlag, pages 500–504, 2002.

[SBCJ02] K.C. Shashidhar, M. Bruynooghe, F. Catthoor and G. Janssens. Geometric Model Checking: An Automatic Verification Technique for Loop and Data Reuse Transformations. In *Proc. of Compiler Optimization meets Compiler Verificaiton (COCV) 2002*, Electronic Notes in Theoretical Computer Science (ENTCS), volume 65, issue 2.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (draft). Technical report, Comp. Sci. Laboratory, SRI International, Menlo Park, CA, 1993.

[Wol99] S. Wolfram. *The Mathematica Book.* Cambridge University Press, 1999.

[ZC91] H. Zima and B. Chapman. *Supercompiler for Parallel and Vector Computers.* Addison-Wesley. 1991.

[ZPG00] L. Zuck, A. Pnueli, and B. Goldberg. Translation Validation of Loop Optimizations in Optimizing Compilers. Technical report, CIMS/CS/NYU. Available in `www.cs.nyu.edu/validator/pubs.html` `http://www.cs.nyu.edu/validator/pubs.html`

[ZPL00] L. Zuck, A. Pnueli, and R. Leviathan. Validations of optimizing compliers. Technical report, Weizmann Institute of Science, 2000.

A Soundness of Validate

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ be two comparable deterministic TSs. Let CP_S and CP_T be the sets of cut-points of P_S and P_T respectively. Assume that the control mapping κ, the data abstraction α, and the invariants φ_is are all defined. Assume further that all verification conditions C_{ij} (for every i and j such that there is a simple path in P_T leading from $\mathtt{Bi}$ to $\mathtt{Bj}$) have been established. We proceed to show that P_T is a correct translation of P_S.

Let

$$\sigma^T : \quad s_1, s_2, \ldots$$

be a finite or infinite computation of P_T, which visits blocks $\mathtt{Bi}_1, \mathtt{Bi}_2, \ldots$, respectively. Obviously $\mathtt{Bi}_1$ is in CP_T and if the computation is finite (terminating) then its last block is also in CP_T. According to an observation made in [Flo67],

σ^{T} can be decomposed into a fusion[3] of simple paths

$$\beta^{T}: \quad \underbrace{(\mathtt{Bj_1}, \ldots, \mathtt{Bj_2})}_{\pi_1^T} \circ \underbrace{(\mathtt{Bj_2}, \ldots, \mathtt{Bj_3})}_{\pi_2^T} \circ \cdots$$

such that $\mathtt{Bj_1} = \mathtt{Bi_1}$, every $\mathtt{Bj_k}$ is in the cut-point set CP^{T}, and the path $\pi_m^{T} = \mathtt{Bj_m}, \ldots, \mathtt{Bj_{m+1}}$ is simple. Since all VCs are assumed to hold, we have that

$$C_{j_k j_{k+1}} \;:\; \varphi_{j_k} \wedge \alpha \wedge \rho^{T}_{j_k j_{k+1}} \quad\rightarrow\quad \exists {V_S}' : \Big(\bigvee_{\Pi \in Paths(\kappa(j_k), \kappa(j_{k+1}))} \rho^{S}_{\Pi} \Big) \wedge \alpha' \wedge \varphi'_{j_{k+1}}$$

holds for every $k = 1, 2, \ldots$.

We can show that there exists a computation of P_S:

$$\sigma^{S}: \quad S_1, \ldots, S_2, \ldots,$$

such that S_1 visits cut-point $\mathtt{B}\kappa(\mathtt{j_1})$, S_2 visits cut-point $\mathtt{B}\kappa(\mathtt{j_2})$, and so on, and such that the source state visiting cut-point $\mathtt{B}\kappa(\mathtt{j_r})$ is compatible with the target state visiting cut-point $\mathtt{Bj_r}$, for every $r = 0, 1, \ldots$.

Consider now the case that the target computation is terminating. In this case, the last state s_r of σ^{T} visits some terminal cut-point $\mathtt{Bj_r}$. It follows that the computation σ^{S} is also finite, and its last state S_m (σ^{T} and σ^{S} are often of different lengths) visits cut-point $\mathtt{B}(\kappa(j_r))$ and is compatible with s_r. Thus, every terminating target computation corresponds to a terminating source computation with compatible final states.

In the other direction, let $\sigma^{S} : S_0, \ldots, S_n$ be a terminating source computation. Let $\sigma^{T} : s_0, s_1, \ldots$ be the unique (due to determinism) target computation evolving from the initial state s_0 which is compatible with S_0. If σ^{T} is terminating then, by the previous line of arguments, its final state must be compatible with the last state of σ^{S}. If σ^{T} is infinite, we can follow the previously sketched construction and obtain another source computation $\widetilde{\sigma}^{S} : \widetilde{S}_0, \widetilde{S}_1, \ldots$ which is infinite and compatible with σ^{S}. Since both S_0 and $\widetilde{S}_0$ are compatible with s_0 they have an identical observable part. This contradicts the assumption that P^{S} is deterministic and can have at most a single computation with a given observable part of its initial state.

It follows that every terminating source computation has a compatible terminating target computation.

B Generating Invariants from Data Flow Analysis

Invariants that are generated by data flow analysis are of definitions that are carried out into a basic blocks by all its predecessors. We outline here the procedure that generates these invariants.

[3] concatenation which does not duplicate the node which appears at the end of the first path and the beginning of the second path

For a basic block B, define:

$$\begin{array}{ll} \texttt{kill}(\texttt{B}): & \text{set of } \texttt{B}\text{'s assignments } y = e(\bar{x}) \\ & \text{some of whose terms are redefined later in } \texttt{B} \\ \texttt{gen}(\texttt{B}): & \text{set of } \texttt{B}\text{'s assignments } y = e(\bar{x}) \\ & \text{none of whose terms are redefined later in } \texttt{B} \end{array}$$

Both kill(B) and gen(B) are easy to construct. VOC-64 constructs two other sets, in(B), that is the set of invariants upon entry to B, and out(B), that is the set of invariants upon exit from B. These two sets are computed by the procedure described in Fig. 13, where *pred*(B) is the set of all blocks leading into B.

```
For every block B
    in(B) init {if BB is initial
                   then set of u = u0 for every variable u
                   else emptyset}
    out(B) init emptyset

repeat for every block B
   out(B) := (in(B) \ kill(B)) ∪ gen(B)
   in(B)  := ⋂_{p∈predB} out(p)
until all sets stabilize
```

Figure 13: Procedure to Compute Invariants

Set operations and comparisons are performed syntactically. An obvious enhancement to our tool is perform those operations semantically. Another possible enhancement is to add inequalities to the computed invariants, which are readily available from the data flow analysis.

VOC-64 computed invariants for both source and target. In optimizations such as code sinking, or even for constant folding together combined with dead code elimination, these invariants are used, though theoretically not necessary.